VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace sloupcově orientovaného úložiště

Column Store Implementation

2015 Miloš Macejech

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Miloš Macejech**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace sloupcově orientovaného úložiště
Column Store Implementation

Zásady pro vypracování:

V aktuálních databázových systémech jsou data nejčastěji uložena po řádcích (celých záznamech). V některých případech je ale vhodné uložit data po sloupcích. Cílem této práce je implementace sloupcově orientovaného úložiště a porovnat jej s úložištěm řádkově orientovaným.

1. Nastudujte možnosti ukládání dat v relačních databázových systémech.
2. Naimplementujte sloupcově orientované úložiště.
3. Proveďte porovnání sloupcově a řádkově orientovaného úložiště a výsledky vyhodnoťte.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Ing. Michal Krátký, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2015

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA 1

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 2. dubna 2015

.....

Rád bych na tomto místě poděkoval vedoucímu mé práce doc. Ing. Michalovi Krátkému Ph.D. za pomoc a rady během konzultací.

Abstrakt

Tato práce se zabývá porovnáním způsobů pro uložení dat v databázových systémech a to řadkově orientovaného ukládání a sloupcově orientovaného ukládání. Řádkově orientované ukládání je rozšířené o index typu B^+-strom a součastí práce je implementace sloupcově orientovaného úložiště a RLE komprimace v tomto úložišti. Tato implementace porovnává různé dotazy pro všechny způsoby ukládání dat a to hlavně podle času vykonání dotazu, počtu I/O operací a komprimačního poměru.

Klíčová slova: sloupcově orientované ukládání, řádkově orienované ukládání, komprimace, RLE, B-strom

Abstract

This work compares ways of storing data in database systems - row-oriented storage and column-oriented storage. The row-oriented storage is extended by B^+-tree index and this work also contains implementation of colum-oriented storage and RLE compression. This implementation compares various queries for each technique by duration of query, I/O operations a compression ratio.

Keywords: column-oriented storage, row-oriented storage, compression, RLE, B-tree

Seznam použitých zkratek a symbolů

ANSI	–	American National Standards Institute
c-store	–	Column store
CPU	–	Central Processing Unit
DBMS	–	Database management system
DDL	–	Data definition language
DML	–	Data manipulation language
MS	–	Microsoft
NULL	–	označení pro žádnou hodnotu nebo nic
I/O	–	Input / Output
IBM	–	International Business Machines Corporation
OLAP	–	Online analytical processing
RAM	–	Random-access memory
RDF	–	Resource Description Framework
RDL	–	Report Definition Language
RLE	–	Run-length Encoding
SEQUEL	–	Structured English Query Language
SŘBD	–	Systém řízení báze dat
SQL	–	Structured Query Language

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato práce se zabývá možnostmi pro uložení dat v databázových systémech. U řádkově orientovaného ukládání, který ukládá data horizontálně se budeme dále věnovat indexům, které se využívají pro optimalizaci výkonu při vyhledávání dat. Nasazení sloupcově orientovaného ukládání přináší časté opakování hodnot ve sloupci, proto se podíváme na možnosti komprimace dat ve sloupcích. Součástí práce je implementace sloupcově orientovaného úložiště, nad kterým provedeme testy a porovnání s ostatními způsoby ukládání dat. Pro přiblížení implementace obsahuje práce několik výpisů důležitých zdrojových kodů se slovním popisem.

Práce je rozdělena do sedmi kapitol. Kapitola 2 představuje databázové systémy. Podíváme se, co je databázový systém, seznámíme se s historií jeho vzniku a tím, jak vypadá dnes. V této kapitole se zaměříme také na jazyk SQL, který byl vyvinut pro manipulaci s daty, definici dat a řízení práv v SŘBD. Další kapitola s pořadovým číslem 3 poukáže na rozdíly řádkově a sloupcově orientovaných databázových systémů. Ukážeme si příklady ukládání dat a jejich hlavní výhody a nevýhody. Ve sloupcově orientovaném ukládání si ukážeme možnosti nasazení a v řádkově orientovaném ukládání si predstavíme B-strom a B^+-strom, který jako index slouží pro zrychlení vyhledávání dat. Součastí představení B^+-stromu je také demonstrace vyhledávání a vkládání. V kapitole 4 se budeme věnovat možnostem komprimace v databázových systémech. Nejprve se podíváme, co je to komprimace a jak se dělí komprimace a nasledně si ukážeme princip několik kódovacích algoritmů a příklady jejich použití. Podrobněji se budeme věnovat kódování RLE.

Na začátku kapitoly 5 si představíme stávající řešení databázového systému RadegastDB[1] a jeho použití. Následuje popis funkcionality již implementovaného řádkově orientovaného ukládání a B^+-strom indexu. Dále se dostaneme k implementaci sloupcově orientovaného ukládání a u tohoto ukládání si ukážeme implementaci komprimace pomocí RLE. Kapitola 6 popisuje prostředí pro testování, kolekci dat pro testování, připravené dotazy a samotné testování. V testech se zaměříme na časy vykonávání dotazů, počet diskových přístupů při vykonávání dotazů a taky na velikost dat na disku v případě komprimace.V závěru kapitoly jsou vyhodnoceny výsledky testů podložené tabulky a grafy.

Závěr 7 shrnuje dosažené výsledky a poznatky získané v této práci.

[1] `http://db.cs.vsb.cz/SubPages/Projects/Projects.aspx`

2 Databázové systémy

Když navštívíte větší webový portál jako vyhledávač (Google, Bing, Yahoo, český Seznam), socialní sít (Facebook, Twitter, LinkedIn) nebo jakýkoliv web, který pracuje s informacemi, je na pozadí databázový systém, který zpracovává všechny uživatelské požadavky. Větší společnosti, které potřebují ukládat různé informace taky využívají databáze pro vyšší efektivitu práce.

Ještě než vznikly první SŘBD, používaly se pro uchovávání informací kartotéky, které bylo možné uspořádat a třídit podle určitých kritérií. Správa kartoték byla ale pro člověka příliš náročná a zdlouhavá, takže byla snaha převést tuto práci na stroje. S velkým pokrokem vývoje počítačů ve druhé polovině 20. století začaly vznikat první databázové systémy. Postupně se objevují pojmy jako SQL, jazyky pro manipulaci a definici dat - DML, DDL.

Existují stovky SŘBD, které vychází z relačního modelu dat, který v roce 1970 poprvé uvedl E.F. Codd [2]. Název tohoto modelu vychází z relační algebry, což je matematický aparát, na kterém relační model dat staví. V tomto modelu jsou údaje uspořádány do tabulek. Tabulka zpravidla shromažďuje údaje o jednom druhu objektů. Můžeme tak mít například tabulku s osobními údaji zaměstnanců. Jednotlivé řádky odpovídají jednotlivým zaměstnancům. Sloupcům tabulky obvykle říkáme v databázové terminologii atributy a jednotlivé řádky se pak nazývají záznamy.

V dnešní době disponují databázové systémy různými funkcemi a silnými nástroji pro práci s daty jako uložené procedury, funkce nebo transakce. Za zmínku určitě stojí zotavitelnost a vysoká dostupnost (záloha dat, clustering). V této oblasti existuje na trhu mnoho řešení ze kterých lze vybírat. Pro menší webový systém bude většinou stačit jednodušší open source databáze MySQL[2]. Naopak pro velkou firmu s nároky na výkon a dostupnost je samozřejmě lepší rešení MS SQL[3] nebo Oracle[4]. Mezi nejznámější používané SŘBD patří dále např. PostgreSQL[5], SQLite[6] [1].

Na SŘBD klademe tyto požadavky: [1]

- umožňuje uživatelům vytvořit novou databázi i tabulky a její schéma (jazyk DDL),
- umožňuje uživatelům dotazovat se na data a měnit data (jazyk DML),

[2] http://www.mysql.com
[3] http://www.microsoft.com/cs-cz/server-cloud/products/sql-server
[4] http://www.oracle.com/cz/database/overview/index.html
[5] http://www.postgresql.org
[6] http://http://www.sqlite.org

- podporuje ukládání velkého objemu dat,
- podporuje databázové transakce, které splňují ACID vlastnosti [1]:
  - trvalost dat - změny, které se provedou jako výsledek úspěšných transakcí, jsou skutečně uloženy v databázi a již nemohou být ztraceny,
  - izolace - přístup k datům od více uživatelů současně bez neočekávané interakce mezi uživateli,
  - atomicida - databázová transakce je jako operace dále nedělitelná. Provede se buď jako celek, nebo se neprovede vůbec,
  - konzistence - transakce převede databázi z jednoho konzistentního stavu do jiného. Transakce tedy zaručuje, že do databáze budou zapsána pouze platná data. Jesliže by mělo dojít k zapsání dat, která porušují integritní omezení, celá transakce bude vrácena zpět a nedojde k žádným změnám.

2.1 Jazyk SQL

Strukturovaný dotazovací jazyk SQL byl vyvinut v 70. letech 20. století ve firmě IBM při výzkumu relačních SŘBD [8]. Nejprv však vznikl jazyk SEQUEL (Structured English Query Language). Cílem bylo vytvořit jazyk, ve kterém by se příkazy tvořily syntakticky co nejblíže přirozenému jazyku (angličtině).

Relační databáze byly stále významnější, a bylo nutné jejich jazyk standardizovat. Americký institut ANSI původně chtěl vydat jako standard zcela nový jazyk RDL. SQL se však prosadil jako de facto standard a ANSI založil nový standard na tomto jazyku. Tento standard bývá označován jako SQL-86 podle roku kdy byl přijat.

V současnosti je mnoho databázových systémů založeno na jazyce SQL, ale implementace nejsou plně prizpůsobené standardu ISO. A naopak, každá z nich obsahuje prvky a konstrukce, které nejsou ve standardech obsaženy. Přenositelnost SQL dotazů mezi jednotlivými SŘBD je proto omezená [1].

Dělení jazyka SQL:

- příkazy pro manipulaci s daty (SELECT, INSERT, UPDATE, DELETE, ...),
- příkazy pro definici dat (CREATE, ALTER, DROP, ...),
- příkazy pro řízení přístupových práv (GRANT, REVOKE),
- příkazy pro řízení transakcí (START TRANSACTION, COMMIT, ROLLBACK),
- ostatní nebo speciální příkazy.

3 Způsoby ukládání dat v SŘBD

Databázové systémy často uchovávají velký objem dat a proto je důležité zvolit nejvhodnejší způsob fyzické implementace SŘBD. Sloupcově orientované databáze ukládájí data do sloupců, zatímco řádkově orientované databázové systémy po řádcích. Každý ze způsobů ukládání má své výhody a nevýhody. Je tedy nutné určit, kdy je který způsob výhodnější nasadit [3].

3.1 Porovnání řádkově a sloupcově orientovaných databázových systémů

Na obrázku 1 můžeme vidět tabulku, která obsahuje seznam produktů a jejich ceny. V řádkově orientovaném databázovém systému je vidět, že záznam/položka je uložena pohromadě na jednom řádku. V sloupcově orientovaném databázovém systému je záznam rozdělen na tolik částí, kolik má tabulka sloupců. Pohromadě jsou tedy uloženy všechny hodnoty prvního sloupce, všechny hodnoty druhého sloupce, atd [3].

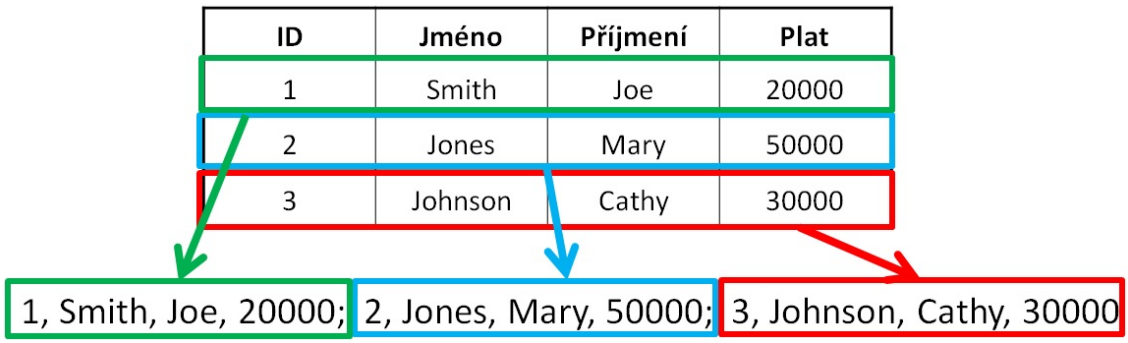
Obrázek 1: Porovnání fyzické implementace SŘBD

3.2 Řádkově orientované ukládání (row-store)

Řádkově orientovaný způsob ukládání dat v podstatě odpovídá rozložení klasické tabulky. Jeden řádek tabulky je uložen pohromadě, za ním další, atd (viz obrázek 2). Řádkově orientované ukádání je navrženo tak, aby systém efektivně vrátil data pro celý řádek s použitím co nejméně operací. Pro zvýšení výkonu podporuje většina SŘBD použití indexů, které ukládají hodnoty vybraných sloupců s ukazateli na řádek, kde se nachází celý záznam entity [3].

Výhody:

- jednoduché vkládání/změna záznamů,
- účinnější při dotazování více atributů najednou.



Obrázek 2: Řádkově orientovaný SŘBD

Nevýhody:

- může číst nepotřebné data,
- slabá účinnost při kompresi oproti sloupcově orientovanému SŘBD.

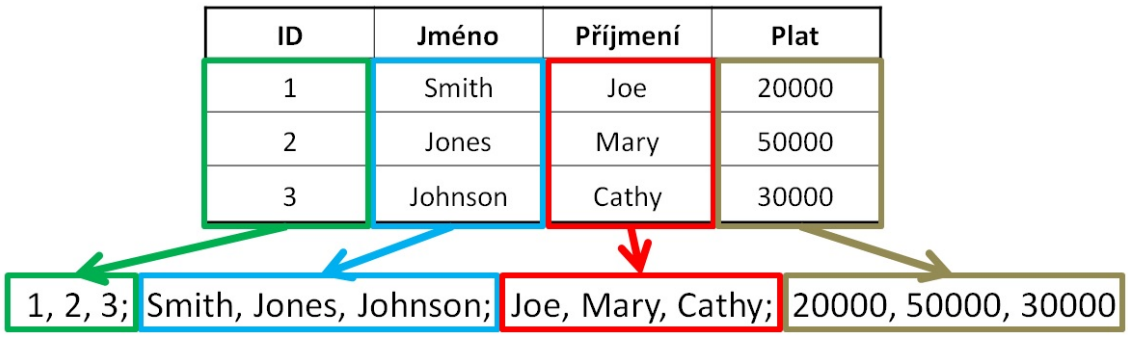
3.3 Sloupcově orientované ukládání (column-store)

Způsob ukládání dat do sloupců je nejvýhodnější, pokud SQL dotaz zahrnuje jen některé sloupce tabulky a selektivita dotazu je nízká. Tím se výrazně sníží počet I/O operací a hlavně čas potřebný pro vykonání dotazu.

Ve sloupcově orientovaném uložení se vždy uloží pohromadě celý sloupec tabulky (viz obrázek 3). Seřazením dat ve sloupci podle hodnot dostáváme velké možnosti pro komprimaci dat, protože se mohou často opakovat stejné hodnoty za sebou. Aby bylo možné spojit jednotlivé atributy entity, vkládá se ke každému atributu unikátní hodnota - číslo řádku. Hodnoty z různých sloupců, které mají stejné číslo řádku, dávájí dohromady logický řádek tabulky [3].

Výhody:

- vysoký výkon u read-only databází, protože se neprovádějí INSERT/UPDATE operace, které jsou náročné z důvodu rozdělení dat sloupců,
- stejné nebo podobné hodnoty jsou uloženy pohromadě, protože ve sloupci jsou data stejného charakteru,
- při práci jen s některými atributy tabulky se výrazně sníží počet přístupů na disk, což je nejnáročnější operace databázového systému, protože přistupujeme jen k datům vybraných sloupců, které navíc můžeme číst sekvenčně



Obrázek 3: Sloupcově orientovaný SŘBD

- vysoký kompresní poměr z důvodu častého opakování hodnot ve sloupci - snížení nákladu na diskovu kapacitu a také snížení počtu I/O (čím méně dat, tím méně I/O),
- účinnější při agregaci nad mnoha záznamy, ale jen pokud se nečtou všechny sloupce,
- změna hodnot v jednom sloupci pro všechny záznamy bez zásahu do ostatních sloupců.

Nevýhody:

- vložení jednoho záznamu do tabulky vyžaduje přístup k více uzlům (pro každý sloupec),
- nišší výkon při dotazování více atributů najednou.
- vkládáním unikátního čísla řádku ke každé hodnotě ve sloupci se zvyšuje celková velikost dat

Nasazení slopcově orientovaného úložište je nejvýhodnější tam, kde převládají SELECT operace a operace INSERT/UPDATE se provádějí ideálně jen v dávkách, aby nezatěžovali systém při čtení. Je to např.: [5]

- Datové sklady - Data Warehousing,
- Dolování dat - Data mining,
- Google Big Table [6],
- Vyhledávání informací - Terabyte TREC[7],
- Vědecká data - SLOAN Digital Sky Survey on MonetDB[8].

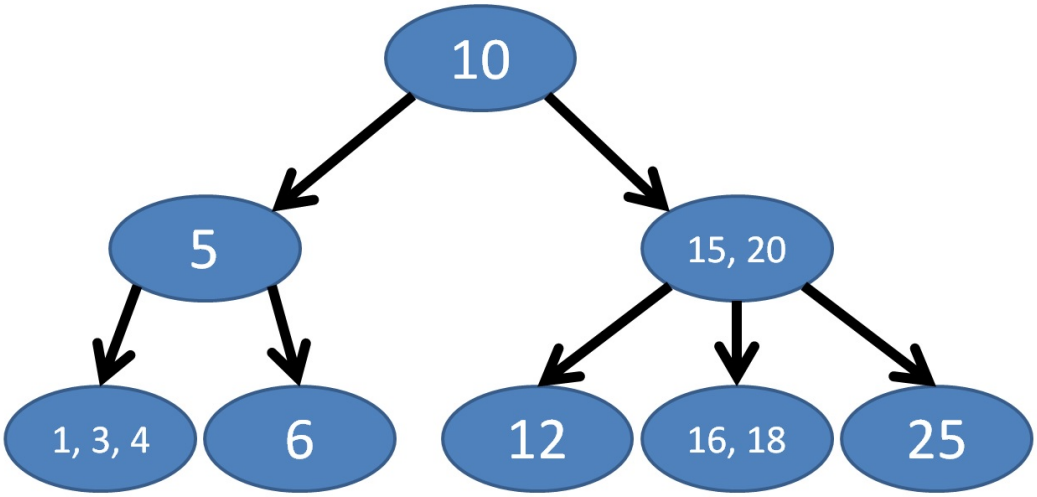
[7] http://www-nlpir.nist.gov/projects/terabyte
[8] https://www.monetdb.org/Home

3.4 B-strom

B-strom je stromová datová struktura, která uchovává setříděné data a umožňuje vyhledávání, vkládání a smazání. Oproti binárnimu stromu umožňuje existenci více než dvou potomků. B-strom je optimalizován pro čtení a zápis velkých bloků dat a proto je hlavně využíván u souborových systémů a databází. Příklad B-stromu můžeme vidět na obrázku 4.

B-strom uvedl Rudolf Bayer v roce 1970. Složitost vyhledávání, mazání a vkládání ve stromu je log $O(\log N)$, kde N je počet položek ve stromu. Protože stránky B-stromu nemusí být vždy naplněny daty dosahuje využití paměti 50% [9].

B-strom představuje velice efektivní strukturu pro uchovávání a vyhledávání hodnot. Její použití je výhodné zejména v případě, že se hodnoty nevejdou do operační paměti a musejí se uchovávat v sekundární paměti - např. na pevném disku. Potom se snažíme omezit na minimum počet přístupů na disk, protože právě přístup na disk je v tomto případě časově nejnáročnější operace [9].



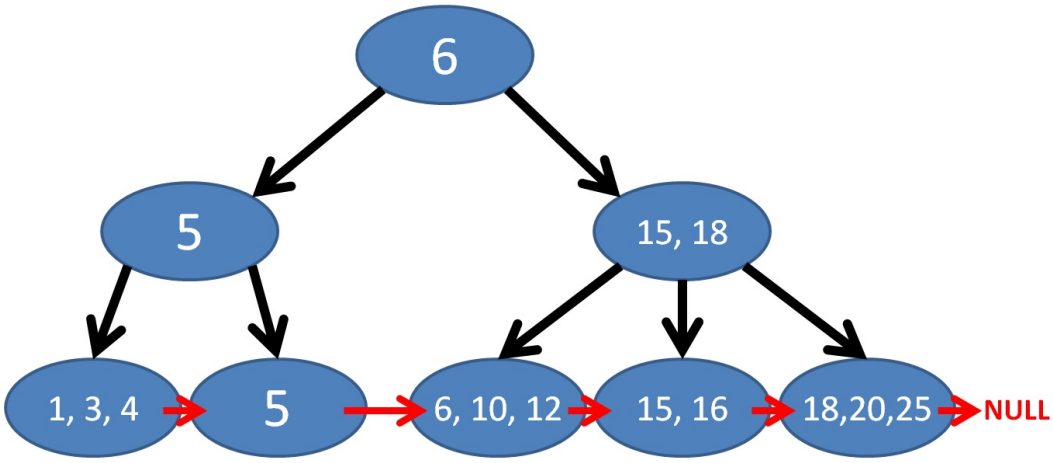
Obrázek 4: B-strom

B-strom řádu n (maximální počet potomků) musí splňovat tyto podmínky:

- všechny uzly, které nemají další potomky musí být na stejné úrovni,
- všechny uzly kromě kořene maji maximálně n a minimálně $[\frac{n}{2}]$ potomků,
- kořen má nejvýše n potomků, minimum je 1

3.5 B$^+$-strom

B$^+$-strom vychází z B-stromu ale na rozdíl od B-stromu má všechny klíče uložené v listech (viz obrázek 5). Ukazatele na klíče jsou však uloženy i ve vnitřních uzlech a v kořenu. Navíc má B$^+$-strom všechny listy spojeny pomocí ukazatele na nasledující list, co umožnuje B$^+$-stromu jak nahodný přístup tak i sekvenční přístup [10].



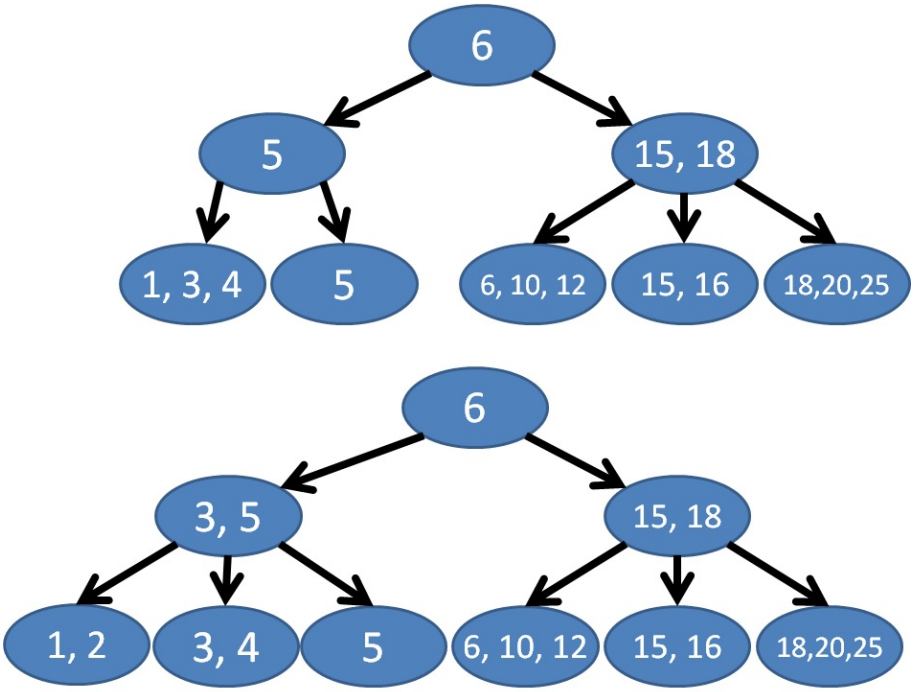
Obrázek 5: B$^+$-strom

Vyhledávání v B$^+$-stromu řádu 4 na obrázku 5 demonstruje nasledující příklad:

Budeme hledat například číslo `16`. Začínáme u kořene, který má jen dva potomky. Číslo `16` je větší než `6`, takže pokračujeme do pravého ulzu `(15, 18)`. Z leva porovnáváme všechny klíče ulzů s vyhledávaným číslem `16`. Protože `16` > `15` a `16` < `18`, tak pokračujeme do uzlu `(15, 16)`. Znova porovnáváme všechny klíče v uzlu až po nalezení čísla `16`. Kdyby číslo `16` nebylo ani v tomto posledním uzlu, tak vyhledávání končí.

Při vkládání do B$^+$-stromu nejprve postupujeme jako při vyhledávání. Pokud již klíč existuje, tak vkládání ukončíme. Pokud neexistuje, pak jej vložíme do listu a odsuneme větší klíče směrem doprava. To ale jen za předpokladu, že v uzlu je volné místo pro nový klíč. Obtížnější situace nastane, když chceme vložit klíč do uzlu, který je úplně plný. V tomto případě ho musíme rozdělit na dva uzly a to tak, že se vybere prostřední klíč z přeplněného uzlu a vytvoří ukazatel v rodiči. Vytvoří se nový uzel, do kterého se přesune prostřední klíč a všechny klíče vpravo od něj. Pokud se ale stane, že vkládáme klíč do rodiče, který je již také plný, rozštěpíme i tento uzel a opět jeden klíč pošleme jeho rodiči. To děláme tak dlouho, dokud nenarazíme na rodiče, ve kterém je volné místo pro klíč. Pokud takto dojdeme až do kořenu a ten je také plný, rozštěpíme kořen a pro nahoru posílaný klíč vytvoříme nový uzel, který se stane novým kořenem obsahujícím jen jeden klíč [9].

Obrázek 6 demonstruje vkládání čísla `2` při maximálním počtu 3 klíčů v uzlu. Vyhledáváním se dostaneme až do uzlu `(1, 3, 4)`, který je již plný. Následuje tedy rozštěpení uzlu na uzly `(1, 2)` a `(3, 4)`. Klíč `3` se vloží do rodiče, který původně obsahoval jen klíč `5`.



Obrázek 6: Vkládání čísla 2

Kvůli podpoře náhodného přístupu nebo rozsahového vyhledávání umožňují databázové systémy vytvářet sekundární indexy (B^+-stromy) pro tabulky. Sekundární index zahrnuje hodnoty indexovaných sloupců a identifikátor příslušného řádku tabulky. Většina komerčních databázových systémů, včetně IBM DB2[9], Sybase Adaptive Server [10], MS SQL Server [11] a Oracle [12] podporuje sekundární indexy pro jejich tradiční tabulky [10].

[9]The SQL Reference. DB2 Universal Database Version 5.2 Publication
[10]Sybase SQL Server, Transact- SQL. (Jser's Guide, Document ID:32300 -01-1100-02, December 1995
[11]Microsoft SQL Server, SQL Server 7.0 Storage Engine, White Paper, October 19980
[12]Oracle8i Concepts, Release 8.1.5: Oracle Corporation, Part Number A67781-01, February 1999

4 Komprimace dat

Komprimace dat je zpracování počítačových dat s cílem zmenšit jejich objem při současném zachování informací v datech obsažených. Obecně se jedná o snahu zmenšit velikost datových souborů, což je výhodné pro jejich archivaci nebo pro přenos přes síť s omezenou rychlostí (snížení doby nutné pro přenos).

Komprimaci dat lze rozdělit do dvou základních kategorií:

- Ztrátová komprimace – při komprimaci jsou některé informace nenávratně ztraceny a nelze je zpět zrekonstruovat. Používá se tam, kde je možné ztrátu některých informací tolerovat a kde nevýhoda určitého zkreslení je bohatě vyvážena velmi významným zmenšením souboru. Používá se pro kompresi zvuku a obrazu (videa), při jejichž vnímání si člověk chybějících údajů nevšimne nebo si je dokáže domyslet (do určité míry).
- Bezeztrátová komprimace – obvykle není tak účinná jako ztrátová komprimace dat. Velkou výhodou je, že komprimovaný soubor lze opačným postupem rekonstruovat do původní podoby. Např. při komprimaci programů, textů, binárních souborů, databází apod.

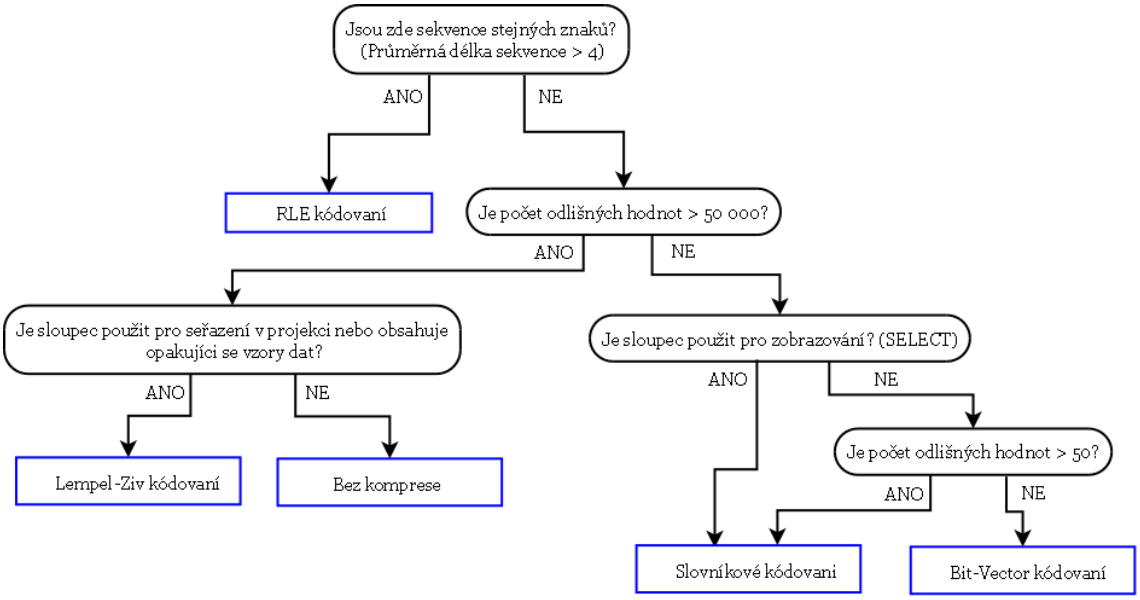
4.1 Komprimace ve sloupcově orientovaném úložišti

Ve sloupcově orientovaných SŘBD se ukládají hodnoty stejného sloupce za sebou a tím vzniká časté opakování stejných hodnot. Toto opakování vytváří vhodné podmínky pro použití komprimace.

Mějme například databázi, která obsahuje informace o zákaznicích (jméno, telefon, e-mail, ulice, město, PSČ, atd.). Uložení těchto dat do sloupců umožňuje uložit všechny jména pohromadě, všechny telefonní čísla pohromadě, atd. Telefonní čísla, ulice, města budou určitě velmi podobné nebo se budou opakovat. Seřazení sloupce navíc umožní kompresi víc hodnot najednou. Komprimace samozřejmě nemá vliv jen na kompresní poměr a tím snížení nároků na diskový prostor. Pokud jsou data komprimována, jsou menší nároky také na pamět a pro načtení dat z disku do paměti tak stačí méně přístupů na disk [4].

4.2 Kódování

Pro komprimaci dat uložených v databázi lze použít více odlišných kódovacích algoritmů, neboli kompresních schémat. Každé kompresní schéma má nejlepší využití v odlišných případech a při splnění určitých podmínek. Publikace Integrating Compression and Execution in Column-Oriented Database Systems [4] uvádí, že při počtu 60 000 000 řádků jejich dosažené výsledky různých kódování vytváří algoritmus, který je možné využít pro rozhodování, které kódování použít pro sloupec. Tento algoritmuts popisuje obrázek 7.



Obrázek 7: Strom popisující rozhodování, které kódování použít [4]

4.3 Potlačení NULL hodnot (NULL Suppresion)

Existuje mnoho variant techniky potlačení NULL hodnot, ale hlavní myšlenka je, že po sobě jdoucí NULL nebo prázdné hodnoty jsou smazány a nahrazeny popisem "kolik jich bylo" a "kde existovali". Tohle kompresní schéma funguje dobře pokud se často vyskytují NULL nebo prázdné hodnoty [4].

4.4 Slovníkové kódování

Slovníkové kódování je pravděpodobně nejrozšířenější schéma komprimace v dnešních databázových systémech [4]. Nahrazuje frekventované hodnoty hodnotami, které mají menší velikost. Algoritmy v této skupině vytvářejí v průběhu komprimace slovník na základě dat již zkomprimovaných, v němž se pak snaží najít data, která se teprve mají komprimovat. Pokud jsou data nalezena ve slovníku, algoritmus zapíše pozici dat ve slovníku místo samotných dat (viz obrázek 8).

Pod slovníkové kódování patří i Lempel-Ziv kódování, které je nejvíc používanou technikou pro beztrátovou souborovou komprimaci. Na tomto algoritmu je založený UNIX-ový komprimační přikaz `gzip`. Hlavní myšlenka algoritmu je, že nahrazuje skupinu znaků, která se již v předchozích datech objevila odkazem na předchozí výskyt [4]. Například komprimace řetězce `"Leze leze po železe"` by vypadala takto: `"Leze l[2,3] po že[5,4]"`

Originální data

LOGIN	Jméno	Příjmení	Telefon	Ulice	Město
7800016	Petr	Novák	731 365 741	Náměstí míru	Ostrava
7800256	Jan	Novák	731 568 453	Náměstí republiky	Praha
7800370	Martin	Pospíšil	731 583 159	17. listopadu	Ostrava
7803846	Pavel	Svoboda	731 479 324	Náměstí míru	Ostrava
7804053	Petra	Nováková	731 528 996	Náměstí svobody	Ostrava

Komprimovaná data

LOGIN	Jméno	Příjmení	Telefon	Ulice	Město
#1 0016	Petr	#2	#3 365741	#4 míru	#5
#1 0256	Jan	#2	#3 568453	#4 republiky	Praha
#1 0370	Martin	Pospíšil	#3 583159	17. listopadu	#5
#1 3846	Pavel	Svoboda	#3 479324	#4 míru	#5
#1 4053	Petra	#2 ová	#3 528996	#4 svobody	#5

Slovníková tabulka

780	#1
Novák	#2
731	#3
Náměstí	#4
Ostrava	#5

Obrázek 8: Slovníkové kódovaní

4.5 RLE (Run-length kódování)

RLE kóduje posloupnost stejných hodnot do dvojic (délka, hodnota). Jedná se o algoritmus, který lze aplikovat na jakýkoliv druh dat. Na jejich charakteru však velice silně závisí výsledný kompresní poměr. Základním principem je sloučit znaky, které se opakují vícekrát za sebou. Tento algoritmus je využíván např. pro komprimaci grafických souborů.

Jako příklad můžeme uvést komprimaci dat `6133331111122222`. Jako identifikátor délky zvolíme např. znak `*`. Číslo `6` se neopakuje, tak se vloží hodnota `*16`. Další číslo je `1`, které se také neopakuje, tak nastává stejná situace - `*11`. Dále máme číslo `3`, které se již ale opakuje 4-krát, vloží se tedy `*43`. Komprimace celého retězce bude vypadat takto: `*16*11*43*51*52`. V případě velikosti jednoho čísla 4 bajty budou mít data před kom-

primací velikost 64 bajtů (16 čísel * 4 bajty). Po komprimaci se velikost změnší na 40 bajtů (10 čísel * 4 bajty).

Při použití RLE komprimace v databázích je důležitý způsob uložení dat. Při řádkově orientovaném ukládání je kódovaní použito hlavně pro komprimaci dlouhých textových atributů, které mají hodně mezer nebo opakujících se znaků [4]. Širší využití je u sloupcově orientovaného ukládání, kde jsou hodnoty atributů uloženy pohromadě a komprimují se hodnoty napříč celým sloupcem. Komprimaci dat ve sloupcově orientovaném ukádání zobrazuje obrázek 9.

Originální data

LOGIN	Jméno	Příjmení	Telefon	Ulice	Město
7800016	Petr	Novák	731 365 741	Náměstí míru	Praha
7800256	Jan	Novák	731 568 453	Náměstí míru	Ostrava
7800370	Martin	Pospíšil	731 583 159	17. listopadu	Ostrava
7803846	Pavel	Pospíšil	731 479 324	Náměstí svodody	Ostrava
7804053	Pavel	Pospíšil	731 528 996	Náměstí svobody	Ostrava
7801395	Pavel	Svoboda	731 469 253	Náměstí svobody	Brno

Komprimovaná data

LOGIN	Jméno	Příjmení	Telefon	Ulice	Město
7800016	Petr	(Novák, 1, 2)	731 365 741	(Náměstí míru, 1, 2)	Praha
7800256	Jan		731 568 453		(Ostrava, 2, 4)
7800370	Martin	(Pospíšil, 3, 3)	731 583 159	17. listopadu	
7803846	(Pavel, 4, 3)		731 479 324	(Náměstí svodody, 4, 3)	
7804053			731 528 996		
7801395		Svoboda	731 469 253		Brno

Obrázek 9: RLE kódovaní

4.6 Bit-Vektor kódování

Bit-Vektor kódování je užitečné, když má sloupec omezený počet možných hodnot. V tomto typu kódování je vytvořen bitový řetězec pro každou unikátní hodnotu, a pomocí bitových hodnot 0/1 se značí jestli na dané pozici je nebo není [4].

Například pro komprimaci dat `113223113321` by se vytvořily 3 bitové řetězce:

- řetězec pro hodnotu 1: 110000110001
- řetězec pro hodnotu 2: 000110000010
- řetězec pro hodnotu 3: 001001001100

V případě komprimace číselných hodnot o velikosti 4 bajty, by měli vstupní data velikost 12 čísel * 4 bajty = 48 bajtů. Komprimace pomocí bit-vektor kódování by zmenšila velikost na 3 řetězce * 12 čísel = 36 bitů = 4,5 bajtů.

5 Vlastní implementace v databázovém systému

5.1 Návrh implementace

Pro porovnávání a testování způsobů ukládání dat je nejprve nutná jejich implementace. Pro implementaci do databázového systému bude použit existující projekt databázového systému RadegastDB[13], který byl na Katedře informatiky VŠB-TU vyvinut. Tento databázový systém pracuje s entitami jako s instacemi třídy `cTuple`. Třída `cTuple` reprezentuje n-rozměrný prvek, který obsahuje pole prvků stejného datového typu. Tato třída vyžaduje pro vytvoření parametr typu `cSpaceDescriptor`, který obsahuje všechna metadata informace o konkrétní instanci třídy `cTuple` jako např. rozměr, který bude v našem případě počet atributů entity a datové typy.

Jako pole entit budeme využívat sekvenční pole `cSequentialArray`, kterému se při definici specifikuje generický typ `cTuple`. Toto sekvenční pole při zavolání metody `Create` naváže spojení s databází QuickDB, přičemž je nutné předat této metodě jako parametr referenci na již existující instanci třídy `cQuickDB`. Pro vytvoření instance třídy `cQuickDB` není vyžadován parametr. Dále je nutné pro vytvoření databáze zavolat metodu `cQuickDB::Create()`, která vyžaduje několik vstupních parametrů:

- **DATABASE_NAME** - název databázového souboru, na disku bude vytvořen soubor .dat + další potřebné soubory s týmto prefixem
- **CACHE_SIZE** - maximální velikost mezipaměti
- **MAX_NODE_INMEM_SIZE** - maximální velikost uzlu v paměti
- **BLOCK_SIZE** - max. velikost jednoho uzlu na disku (v našem případě 8192 B)

Řádkově orientované úložiště tedy nevyžaduje žádnou další implementaci. Stačí jedno sekvenční pole do kterého budeme vkládat entity jako prvek typu `cTuple`. Entity budou tedy vloženy vedle sebe a vzniká tak řádkově orientované ukládání. Jako reprezentace řádkově orientovaného ukládání byla vytvořena třída `cRowsStoreArray` s kontruktorem, který očekává parametr `int attrLen` - počet atributů entity.

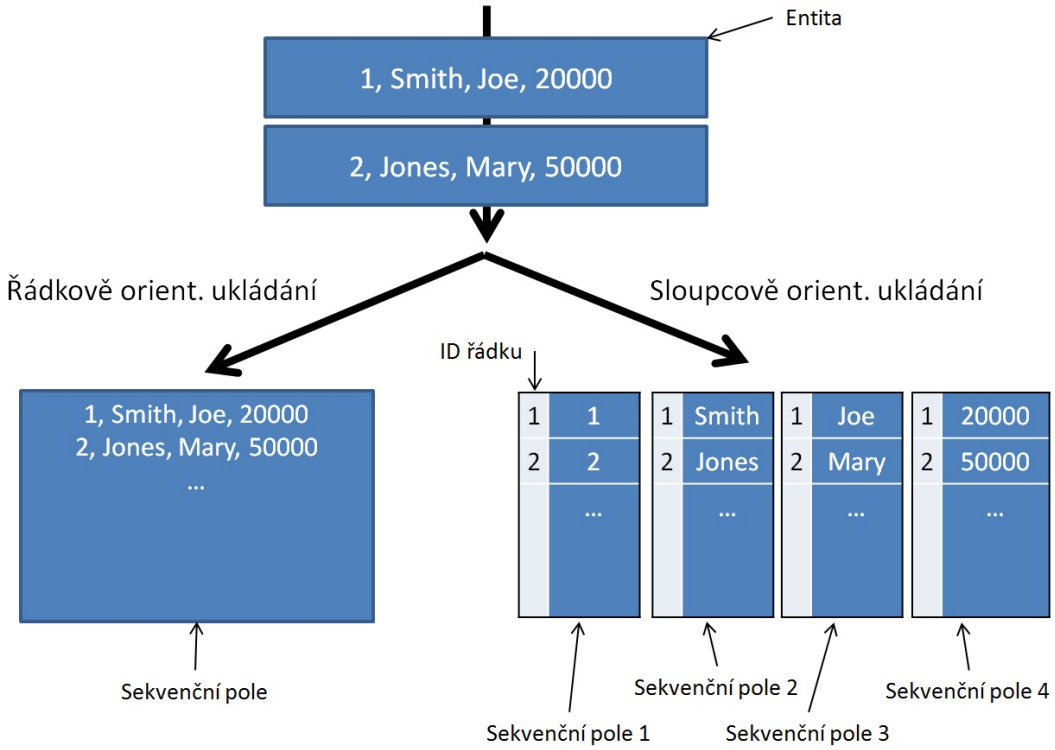
Pro výhledávání dat v tabulce slouží třída `cSequentialArrayContext` a její metody `OpenContext` (otevře a nastaví na první položku), `Advance` (posunutí kurzoru na další prvek v poli) a `CloseContext` (uzavření). Hodnoty atributů získáme pomocí statické metody třídy `cTuple::GetUInt()`, které mimo jiné předáme jako parametr index sloupce požadované hodnoty.

Pro zrychlení vyhledávání v řádkově orientovaném ukládání jsme zvolili index typu B^+-strom. Implementaci ukládání dat s tímto indexem reprezentuje třída `cRowStoreBTreeArray`. Hlavní logika B^+-stromu se nachází v již existující třídě `cBpTree`,

[13] `http://db.cs.vsb.cz/SubPages/Projects/Projects.aspx`

která pro vytvoření vyžaduje hlavičkový objekt typu `cBpTreeheader`. Pomocí tohoto objektu musíme povolit vkládání duplicitních klíču do pole. Slouží k tomu metoda `DuplicatesAllowed()`, které předáme parametr `true`. Vyhledávání v B^+-stromu nám zajišťuje metoda `PointQuery()`, která vyhledává podle parametru typu `cTuple`.

Pro sloupcově orientované úložiště musíme naimplementovat ukládání tak, že se budou ukládat vedle sebe hodnoty sloupce a toho docílíme tak, že pro každý atribut vytvoříme samostatné sekvenční pole, do kterého budeme vkládat jen hodnoty jednoho atributu. Tyto hodnoty tak budou fyzicky oddělené od hodnot jiných atributů. Toto rozdělení do více sekvenčních polí demonstruje obrázek 10).



Obrázek 10: Sekvenční pole

Ve sloupcově orientovaném ukládání budeme dále implementovat komprimaci dat pomocí RLE kódování, protože na rozdíl od řádkově orientovaného ukádání se stejné nebo podobné hodnoty ukládájí za sebou, a tak jsou vytvořeny vhodné podmínky pro nasazení tohoto kódování. RLE kódování bude do sloupcově orientovaného úložiště implementováno s použitím stávajícího sekvenčného pole. Při vkládání do pole proběhne komprimace a selekt operace můžou být podle situace provedeny nad daty komprimovanými nebo se provede dekomprimace. RLE kódování komprimuje data do dvojic (hodnota, počet opakování) a abychom mohli data zpětně dekomprimovat, musíme rozlíšit,

zda se jedná o hodnotu nebo počet opakování. Pro počet opakování tedy budeme nastavovat první bit na 1 (viz obrázek 11).

Obrázek 11: RLE bitové označení

5.2 Implementace sloupcově orientovaného úložiště

Implementace sloupcově orientovaného ukládání se nachází v třídě `cColumnStoreArray`. Na rozdíl od řádkově orientovaného ukládání je nutné vytvořit tolik sekvenčních polí, kolik má tabulka atributů. Vytvoříme tedy místo jednoho objektu (`cSequentialArray, cSequentialArrayHeader, cSequentialArrayContext`) pole objektů těchto typů. Vytvoření popisuje výpis kódu 1.

```
1  mContext = new cSequentialArrayContext<cTuple>*[attrLen];
2  mHeader = new cSequentialArrayHeader<cTuple>*[attrLen];
3  mSeqArray = new cSequentialArray<cTuple>*[attrLen];
4  for (int i = 0; i < attrLen; i++)
5  {
6      // headers
7      stringstream headerName;
8      headerName << "seqArray" << i;
9      string tmp = headerName.str();
10     mHeader[i] = new cSequentialArrayHeader<cTuple>(tmp.c_str(), BLOCK_SIZE, mSpaceDsc,
           cDStructConst::DSMODE_DEFAULT);
11     mHeader[i]->SetCodeType(ELIAS_DELTA);
12     // sequential array
13     mSeqArray[i] = new cSequentialArray<cTuple>();
14     mContext[i] = new cSequentialArrayContext<cTuple>();
15
16     if (!mSeqArray[i]->Create(mHeader[i], quickDB))
17     {
18         printf ("Sequential array: creation failed\n");
19     }
20 }
```

Výpis 1: Vytvoření sloupcově orientovaného úložiště

Složitější tak bude i vkládání prvků, protože jedna entita se musí rozdělit a vložit do několik sekvenčních polí. Pro metodu `Insert()` jsme pro jednotnost všech způsobů ukádání dat zvolili parametr `cTuple`. Musíme tedy z jednoho tohoto objektu vytvořit tolik objektů, kolik má entita atributů. Vkládání prvků popisuje výpis kódu 2.

```
1  void cColumnStoreArray::Insert(cTuple *tuple)
2  {
3      cTuple *t = new cTuple(mSpaceDsc, 1);
4      for (int i = 0; i < attributesLen; i++)
5      {
6          unsigned int currentValue = tuple->GetUInt(i, mSpaceDscAll);
7          // insert
8          t->SetValue(0, currentValue, mSpaceDsc);
9          mSeqArray[i]->AddItem(nodeid, position, *t);
10     }
11     delete t;
12 }
```

Výpis 2: Vkádání do sloupcově orientovaného úložiště

5.3 Implementace RLE komprimace ve sloupcově orientovaném úložišti

Sloupcově orientované komprimované pole pomocí RLE kódování v našem projektu najdeme pod názvem třídy `cColumnStoreCompressedArray`. Oproti implementaci bez komprimace obsahuje pole `unsingned int lastInserted` entity, které slouží pro zapamatování poslední vložené hodnoty do sloupce a další pole o stejné velikosti pro počítání počtu opakování vložených hodnot do sloupce.

Při opakování hodnot je v RLE kódování nutné vložit v páru dvojici (počet opakování, hodnota). Abychom mohli rozlišit, zda se jedná o hodnotu nebo o počet opakování, budeme hodnotu počtu opakování označovat pomocí prvího bitu a to tak, že `0` značí hodnotu a `1` značí počet opakování. Pro bitové operace byly vytvořeny nasledující metody:

- `MarkAsLengthValue` - označí jako počet opakování (první bit = 1)
- `CheckForLengthValue` - kontroluje první bit (0 = false, 1 = true)
- `GetRealValue` - vrátí realní hodnotu v případe počtu opakování (první bit = 0)

Vkládání (komprimace) dat

Pokud je vkládaný prvek první v tabulce, provede se okamžité vložení (viz řádek 15). Pokud již nějaký prvek byl vložen, provede se kontrola opakování hodnoty (řádek 22).

Pokud se jedná o opakování (aktuálně vkládaná hodnota je stejná jako poslední vkládaná), navýší se počet opakování v poli pro příslušný sloupec (řádek 24). Když je vkládaná hodnota jiná než předchozí, provede se dále kontrola počtu opakování poslední hodnoty (řádek 28) a v případě, že se jedná o opakování, se tento počet vloží do databáze s nastaveným prvním bitem na `1` (řádek 35). Následně se vloží aktuálně vkládaná hodnota. Tento algoritmus popisuje výpis kódu 3.

Problém při tomto vkládání nastane ve chvíli, když bude poslední vložená hodnota stejná jako ta předchozí, protože vložení počtu opakování probíha až po vložení hodnoty, která je jiná jako předchozí. Toto jsme vyřešili pomocí metody `FinishInsert()`, která provede vložení všech počtů opakování na konec sloupce. Tuto metodu není nutné volat po každem vkládání dat. Pokud se vkládá více dat po sobě, jde o tzv. dávku, stačí zavolat tuto metodu po posledním vložení.

```
1  bool cColumnStoreCompressedArray::Insert(cTuple *tuple)
2  {
3      cTuple *tValue = new cTuple(mSpaceDsc, 1);
4      cTuple *tLength = new cTuple(mSpaceDsc, 1);
5
6      for (int i = 0; i < attributesLen; i++)
7      {
8          unsigned int currentValue = tuple->GetUInt(i, mSpaceDscAll);
9          int runLengthsTotalCounter = 0;
10
11         if ( isFirstInsert ) // prvni se vzdy vlozi
12         {
13             // insert
14             tValue->SetValue(0, currentValue, mSpaceDsc);
15             mSeqArray[i]->AddItem(nodeid, position, *tValue);
16
17             compressedCount[i]++;
18         }
19         else
20         {
21             unsigned int lastValue = lastInserted[i];
22             if (lastValue == currentValue) // stejna hodnota jak predesla
23             {
24                 runLength[i]++;
25             }
26             else
27             {
28                 if (runLength[i] > 1) // vlozeni runlength pred dalsi hodnotou
29                 {
30                     unsigned int runLenghtValue = runLength[i];
31                     MarkAsLengthValue(runLenghtValue); // nastaveni prvniho bitu na 1
32
33                     // insert
34                     tLength->SetValue(0, runLenghtValue, mSpaceDsc);
35                     mSeqArray[i]->AddItem(nodeid, position, *tLength);
36
```

```
37                      runLength[i] = 1; // vychozi hodnota
38                  }
39                  // insert
40                  tValue->SetValue(0, currentValue, mSpaceDsc);
41                  mSeqArray[i]->AddItem(nodeid, position, *tValue);
42
43                  compressedCount[i]++;
44              }
45          }
46          lastInserted[i] = currentValue;
47      }
48      isFirstInsert = false;
49
50      delete tValue;
51      delete tLength;
52
53      return true;
54  }
```

Výpis 3: RLE Vložení dat / komprimace

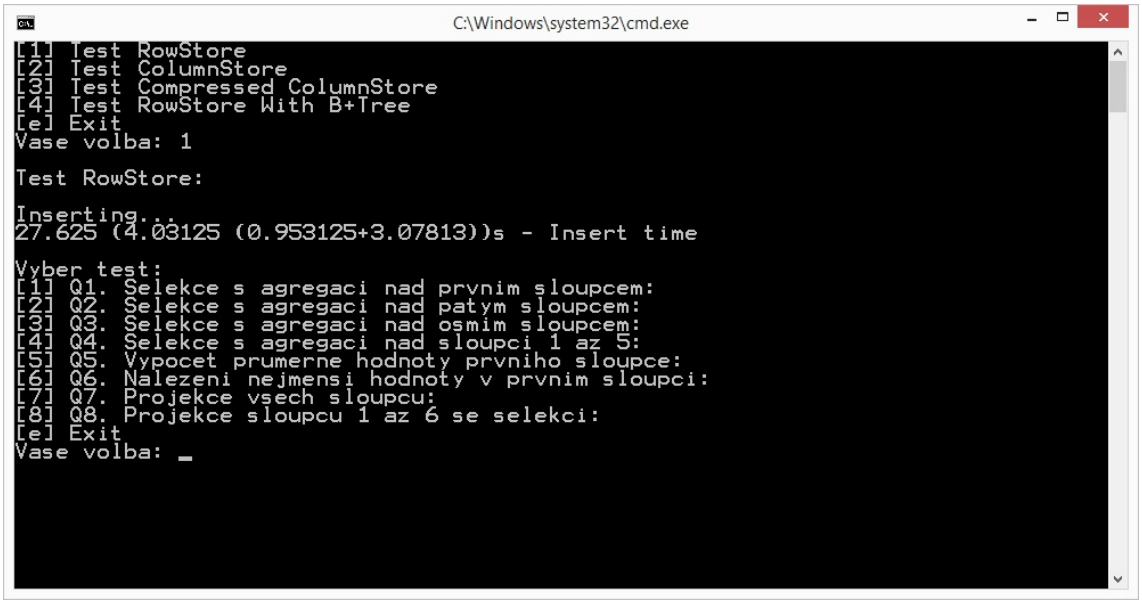
6 Porovnání způsobů ukládání dat

Pro testování všech způsobů ukládání dat bylo vytvořeno několik testovacích dotazů, které budeme porovnávat nad velkou kolekcí dat. Zaměříme se např. na čas vkládání dat, který podle získaných informací očekáváme menší u řádkově orientovaného ukládání. Select operace nad jedním sloupcem by zas měly být rychlejší u sloupcově orientovaného úložiště. Zajímavé výsledky také očekáváme od použití B^+-strom indexu u řádkově orientovaného ukládání. Komprimace pomocí RLE nám určitě přinese zrychlení a menší objem dat.

6.1 Vybavení hardware a aplikace pro testování

Všechny testy budou spušteny na stejném počítači s dvoujádrovým procesorem Intel Core i3-4130 @ 3.4GHz, pamět RAM 8GB, obyčejný pevný disk 7200rpm. Operační systém této sestavy je Windows 8.1 Pro x64. Pro vývoj aplikace bylo použité vývojové prostředí od Microsoftu Visual Studio 2013 a programovací jazyk C++.

Aplikace byla zkompilována v režimu `release` a na pozadí operačního systému běželi jen systémové procesy. Konzolová aplikace umožnuje vybrat požadovaný test ze seznamu pro každý způsob uložení dat, viz obrázek 12.



Obrázek 12: Konzolová aplikace

6.2 Kolekce dat

Testovací data představuje textový soubor `dataset.txt`[14], který obsahuje 4 999 999 řádků. Každý řádek má 12 číselných hodnot oddělených čárkou. Načítání těchto dat do dvou-rozměrného pole aplikace zajišťuje statická třída `cTestCollection` a její metoda `fileopen`. Charakter dat pro vybrané sloupce použité pro testování popisuje tabulka 13.

sloupec	počet různých hodnot
sloupec 0	1823
sloupec 1	19982
sloupec 2	5
sloupec 3	250
sloupec 4	1000
sloupec 5	50000
sloupec 6	10000
sloupec 7	4999999
sloupec 8	10
sloupec 9	595
sloupec 10	300
sloupec 11	296

Obrázek 13: Testovací data

6.3 Testovací dotazy

Celkem bylo vytvořeno 8 testovacích dotazů:

Q1. Selekce s agregací nad prvním sloupcem

Pvní sloupec obsahuje hodnoty, které se opakují velmi často. Z celkového počtu `4 999 999` řádků je duplicitních `4 996 127` . Dotaz bude vyhledávat číslo `1492` , které nalezne 3933-krát. U toho dotazu očekáváme kvůli častému opakování vysoký výkon u column-store databáze komprimované pomocí RLE kódování.

Zápis v jazyce SQL: `SELECT COUNT(*) FROM table WHERE column0 = 1492`

[14] `https://goo.gl/su2WJJ`

Q2. Selekce s agregací nad pátým sloupcem

V tomto sloupci se hodnoty opakují zřídka. Duplicitních hodnot je přibližně 70%. Dotaz vrátí `4957` shod pro podmínku vyhledávání čísla `590`.

Zápis v jazyce SQL: `SELECT COUNT(*) FROM table WHERE column4 = 590`

Q3. Selekce s agregací nad osmým sloupcem

Tento sloupec obsahuje všechny hodnoty unikátní. Podmínka po vyhledáván je číslo `2711340`.

Zápis v jazyce SQL: `SELECT COUNT(*) FROM table WHERE column7 = 2711340`

Q4. Selekce s agregací nad sloupci 1 až 5

Dotaz v tomto testu vyhledává záznam, který splňuje 5 podmínek (pro každý sloupec jedna). Počet výsledků: `1`.

Zápis v jazyce SQL: `SELECT COUNT(*) FROM table WHERE column0 = 590 AND column1 = 16209 AND column2 = 1 AND column3 = 181 AND column4 = 292`

Q5. Výpočet průměrné hodnoty prvního sloupce

Agregace nad sloupcem s výpočtem průměrné hodnoty.

Zápis v jazyce SQL: `SELECT AVG(column0) FROM table`

Q6. Nalezení nejmenší hodnoty v prvním sloupci

Agregace nad sloupcem s nalezením nejmenší hodnoty.

Zápis v jazyce SQL: `SELECT MIN(column0) FROM table`

Q7. Projekce všech sloupců

Dotaz vypíše všechna data tabulky.

Zápis v jazyce SQL: `SELECT * FROM table`

Q8. Projekce sloupců 1 až 5 se selekcí

Dotaz vypíše atributy 1 - 5 entity, která splňuje podmínku `column3 = 181`

Zápis v jazyce SQL: `SELECT column0, coulumn1, column2, column3, column4 FROM table WHERE column3 = 181`

6.4 Výsledky testování

Pří vkládání dat do databází jsme hli podobné výsledky u všech způsobů uložení dat. Nejpomalejší bylo sloupcově orientované úložiště, protože pri vkládání musí entitu rozdělit do více sloupců. Nejlepší čas, a tedy rychlejší než řádkově orientované úložiště, bylo sloupcově orientované úložiště s RLE komprimací a to z toho důvodu, že dat bylo po komprimaci méně a proběhlo tím pádem méně vložení do databáze. Komprimace zmenšila velikost dat na 70% původní velikosti. Kompletní časy vkládání dat a velikost datového souboru databáze po naplnění daty zobrazuje tabulka 14.

	row-store	column-store	column-store RLE
velikost dat	235 112kB	235 112kB	169 088kB
čas vkládání	28.5s	33.6s	25.7s

Obrázek 14: Vkládání dat

Dotazy pro průchod jedním sloupcem (testy Q1 - Q3) byly rychlejší přibližně o 50% u sloupcově orientovaného úložiště oproti řádkově orientovanému úložišti. Test Q4, který prochází 5 sloupců, byl taky rychlejší u sloupcově orientovaného úložiště, ale rozdíl již nebyl tak velký. Agregační testy (Q5, Q6) dopadli podobně jako testy Q1 - Q3. U testu Q7, který čte všechna data tabulky, můžeme vidět přibližně stejný čas a počet diskových přístupů u sloupcově i řádkově orientovaného úložiště. Z celkového porovnání výsledků dotazů pro řádkově a sloupcově orientovaného úložiště je vidět, že u řádkově orientovaného úložiště jsou hodnoty času a diskových přístupů u všech testech podobné, zatímco u sloupcově orientovaného úložiště jsou výsledky zavíslé na počtu sloupcích, se kterými pracuje testovací dotaz.

Pro sloupcově orientované úložiště komprimované pomocí RLE kódování je u testech Q1 - Q3 pozoruhodné, jak se mění časy podle toho, jak často se opakují hodnoty ve sloupci. Při velmi častém opakování (test Q1) se čas snížil na 0,58% původního času bez komprimace, ale pro sloupec kde se žádné hodnoty neopakovali (test Q3) zůstal čas přibližně stejný, jako bez komprimace. Agregační testy nad jedným sloupcem (Q5, Q6) dosáhly výsledky porovnatelné s testem Q1, protože se nemusela provádět dekomprimace celého sloupce. Nepříznivé výsledky s použítím komprimace dosáhly testy Q4, Q7,

Q8, které pracují s více sloupci, a pro jejich spojení do logického řádku byla potřebná dekomprimace dat.

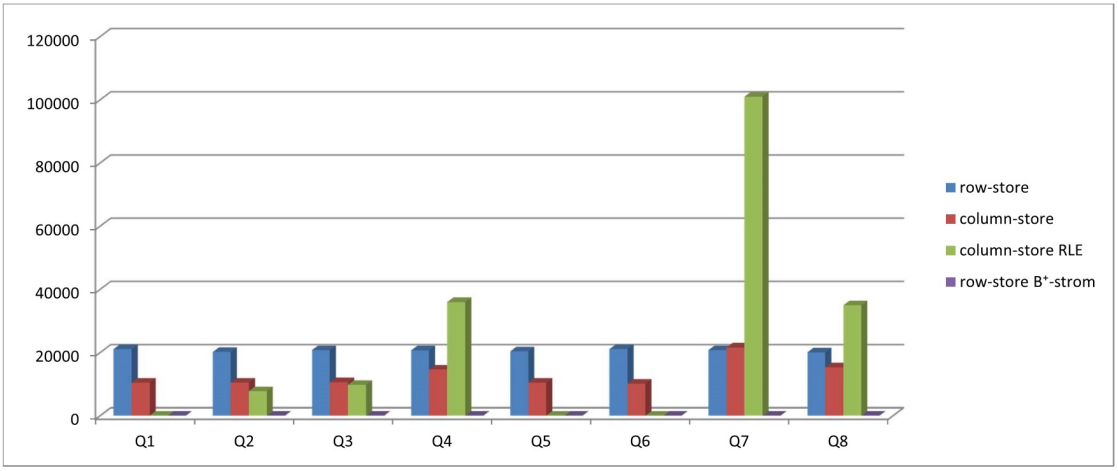
S použitím a B^+-strom indexu u řádkově orientovaného úložiště nad sloupcem použitém pro selekci v testech Q1 - Q3 jsme získali nejlepší časy vykonání dotazů a to v řádech desítek milisekund. Pro agregační testy by použití indexu nemělo smysl, protože se agregace provádí nad celou tabulkou a je nutný průchod všemi řádky tabulky. Všechny naměřené časy trvání dotazů a také počty diskových přístupů zobrazují tabulky 15, 16 a grafy 17, 18.

test	row-store	column-store	column-store RLE	row-store B^+-strom
Q1	21100	10500	70	**10**
Q2	20200	10500	7800	**15**
Q3	20800	10600	9800	**17**
Q4	20700	**14600**	36000	-
Q5	20400	10500	**70**	-
Q6	21100	10200	**20**	-
Q7	**20800**	21600	101000	-
Q8	20100	15300	35000	-

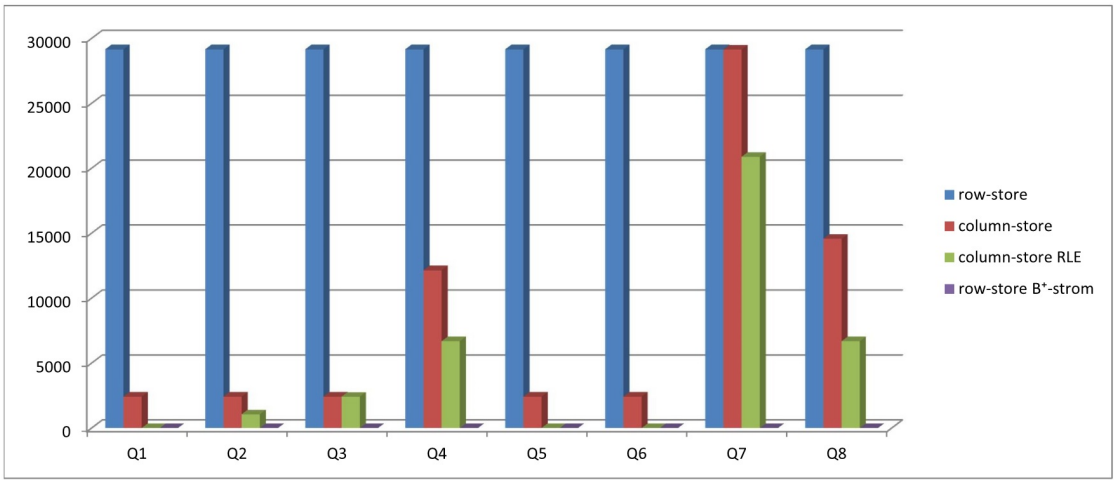
Obrázek 15: Výsledky dotazů podle času [ms]

test	row-store	column-store	column-store RLE	row-store B^+-strom
Q1	29162	2428	**3**	8
Q2	29162	2428	1061	**9**
Q3	29162	2428	2420	**3**
Q4	29162	12141	**6684**	-
Q5	29162	2428	**12**	-
Q6	29162	2428	**3**	-
Q7	29162	29139	**20883**	-
Q8	29162	14569	**6684**	-

Obrázek 16: Výsledky dotazů podle počtu diskových přístupů



Obrázek 17: Graf výsledků podle času [ms]



Obrázek 18: Graf výsledků podle počtu diskových přístupů

7 Závěr

V této práci jsme porovnali možnosti ukládání dat a to řádkově orientovaného ukládání a sloupcově orientovaného ukládání. Nejzásádnější rozdíl je ve fyzické implementaci, kde řádkově orientované ukládání ukládá data horizontálně a sloupcově orientované ukládání vertikálně. Ukázali jsme si výhody a nevýhody obou způsobů ukádání dat a v které oblasti je nejvýhodnější jejich nasazení.

Provedené testy nad našimi implementovanými systémy potvrdily předpoklady z úvodu této práce. Vkládání dat je rychlejší u řádkově orientovaného ukládání, ale na druhou stranu při použití RLE kódování a zároveň častém opakování hodnot u sloupcově orientovaného ukládání může být čas vkládání dat ještě rychlejší jako u řádkově orientovaného ukládání. Sloupcově orientované úložiště prokázalo nejlepší výsledky při práci s jedním nebo menším počtem atributů tabulky.

Po vytvoření indexu v řádkově orientovaném úložišti na sloupci použitém v selekci jsme dostali nejlepší časy zpracování dotazů. Indexy mohou být velmi užitečné, ale musíme myslet na to, že index zabírá dašlí místo v paměti, které může být u velkých tabulek větší jako data samotná. Při vkládání a změně dat je nutné provést změny také v indexu a tím se zpomalují tyto operace. Pro správné fungování indexů je navíc důležité vědět, jaké dotazy jsou používané nad tabulkou a jaká data tabulka obsahuje. Při špatném vytvoření indexů, nebo vytvoření příliš mnoho indexů, je jejich vliv na zpracování dat spíše negativní.

Komprimace v databázových systémech přináší značné výhody a námi zvolené kódování RLE ve sloupcově orientovaného ukládání nám to potvrdilo. Hlavním faktorem pro dosažení nejlepších výsledků komprimace, zmenšení objemu dat, je u RLE kódování opakování stejných hodnot za sebou. V našem případě se zmenšila velikost dat na 70% původní velikosti. V případě výskytu více stejných hodnot by byly výsledky ještě lepší. Komprimace přináší také ekonomické výhody. Pokud by databáze za 10 let narostla do velikosti 100 GB a dostala se tak na hranici kapacity diskového pole, tak s použitím komprimace můžeme provozovat databázi několik dalších let bez rozširování kapacity.

Shrnutím výsledků můžeme jednoznačně potvrdit, že sloupcově orientované ukládání má mezi databázovými technologiemi zasloužené místo a uplatnění si najdou hlavně u větších datových skladů pro analýzu dat. Tohoto jsou si vědomi i vývojáři nejvýznamnějších databázových systémů, kteří již začali poskytovat sloupcově orientované ukádání ve svých databázích.

Miloš Macejech

8 Reference

[1] GARCIA-MOLINA, Jeffrey D ULLMAN, Hector a WIDOM, Jennifer., *Database Systems: The Complete Book. 2nd ed.* New Jersey: Prentice Hall, 2002, 1119 s. ISBN 01-303-1995-3.,

[2] DAFOULAS, Patricia Ward and George. *Database management systems.* London: Thomson, 2006. ISBN 9781844804528.

[3] J. Abadi, S. Madden, N. Hachem, *Column-Stored vs. Row-Stores: How Different Are They Really?*, New York, 2008., http://db.csail.mit.edu/projects/cstore/abadi-sigmod08.pdf

[4] Daniel J. Abadi , Samuel R. Madden , Miguel C. Ferreira, *Integrating Compression and Execution in Column-Oriented Database Systems*, New York, 2006., http://db.csail.mit.edu/projects/cstore/abadisigmod06.pdf

[5] Daniel Abadi, Peter Boncz, and Stavros Harizopoulosa, *VLDB 2009 Tutorial on Column-Stores*, 2009., http://www.slideshare.net/abadid/vldb-2009-tutorial-on-columnstores

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, 2006., http://static.googleusercontent.com/media/research.google.com/cs//archive/bigtable-osdi06.pdf

[7] Thomas Kejser, *How Do Column Stores Work?*, 2012., http://kejser.org/how-do-column-stores-work/

[8] Don Chamberlin, *SQL*, San Jose, CA http://researcher.ibm.com/researcher/files/us-dchamber/SQL-encyclopedia-entry.pdf

[9] Michal Krátký, Jiří Dvorský, *Úvod do programování*, http://www.cs.vsb.cz/kratky/courses/2004-05/udp/presentation/udp-10_6.pdf

[10] Amruta Kudale, *B+ tree Preference over B Tree*, Chicago, USA http://www.academia.edu/11575258/B_tree_preference_over_B_trees

9 Seznam příloh na CD/DVD

…

```
root
├── BakalarskaPrace.pdf – Bakalářská práce
└── src
    └── test
        └── mbt
            └── mac0172
                └── compression – zdrojový kód
                    └── dataset.txt – kolekce testovacích dat
```